Naval Research Laboratory

Washington, DC 20375-5320

NRL/FR/5510--95-9777

Extending the User Action Notation (UAN) for Specifying Interfaces with Multiple Input Devices and Parallel Path Structure

LYNN A. DIEVENDORF, JR.
DEREK P. BROCK, JR.

Human Computer Interaction Laboratory
Naval Center for Applied Research in Artificial Intelligence
Information Technology Division

ROBERT J. K. JACOB

Tufts University
Medford, Massachusetts

May 12, 1995

Approved for public release; distribution unlimited.

REPORT DOCUMENTATION PAGE	Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY (*Leave Blank*)

2. REPORT DATE
May 12, 1995

3. REPORT TYPE AND DATES COVERED

4. TITLE AND SUBTITLE
Extending the User Action Notation (UAN) for Specifying Interfaces with Multiple Input Devices and Parallel Path Structure

5. FUNDING NUMBERS
PE - 0601153N
TA - R5391
SC - 019501

6. AUTHOR(S)
Lynn A. Dievendorf Jr., Derek P. Brock Jr., and Robert J.K. Jacob*

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Research Laboratory
Washington, DC 20375-5320

8. PERFORMING ORGANIZATION REPORT NUMBER
NRL/FR/5510--95-9777

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Office of Naval Research
Arlington, VA 22217-5000

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

Accesion For		
NTIS	CRA&I	X
DTIC	TAB	☐
Unannounced		☐
Justification		
By		
Distribution /		
Availability Codes		
Dist	Avail and / or Special	
A-1		

11. SUPPLEMENTARY NOTES
*Tufts University, Medford, Massachusetts

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (*Maximum 200 words*)

Many modern computer applications provide the user with a choice of several different methods of completing the same task. These options are afforded the user by the implementation of multiple input devices (e.g., a mouse and a keyboard) and keyboard shortcuts, or hot-keys, into the interface. Although these options can be described by current interface specification methods, there is a need for a notation that can clearly represent this type of interface design in a meaningful format. A survey of formal specification methods was conducted in an attempt to identify a method which captured these elements of the human-computer interface. The User Action Notation (UAN) shows promise for illustrating these options and providing an analytical tool for interfaces with this type of parallel structure. A graphing task from a DOS application called SigmaPlot™ is then specified in UAN. During this process, structural modifications to the standard UAN format are made, and new symbols are added to the existing repertoire in an effort to augment the notation's analytical strengths. The modifications prove to clarify the specification of interfaces with multiple input devices and allow the reader of the specification to make meaningful comparisons of alternative methods for completing a specified task. These modifications are documented, and their benefits are discussed.

14. SUBJECT TERMS
Specification methods
User action notation
Interface design
Human-computer interaction
Usability

15. NUMBER OF PAGES
38

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std 239-18
298-102

CONTENTS

EXTENDING THE USER ACTION NOTATION (UAN) FOR SPECIFYING INTERFACES WITH MULTIPLE INPUT DEVICES AND PARALLEL PATH STRUCTURE

1. INTRODUCTION

Software developers, driven by competition and realizing the importance of the user interface, are shifting toward more iterative design methods. This process allows for more user involvement in the design process and "is necessary to arrive at a successful system" (Shneiderman 1987). When usability is a primary concern, it is essential that the developers have a specification technique that can clearly and accurately specify the user interface and the user actions necessary to navigate the interface. Such specification methods are invaluable to the developer for identifying inconsistencies in the interface and analyzing data collected in pilot studies on the usability of prototype applications. Until recently notations suitable for such specification have fallen short. "Formal and informal specification techniques have been applied to many aspects of software systems. However, specification techniques have been less successful in describing the interface between the system and the user" (Jacob 1986). There now exist many specification techniques capable of describing user actions and the user interface. However, when describing complex tasks with many alternate ways of performing the same action (e.g., via two or more input devices), these specifications become quite confusing and fail to clearly represent this parallel structure.

The present work was inspired by an experiment conducted by Schmidt-Nielsen and Ackerman (1993) investigating individual differences in user strategies during repetitive task situations. A DOS application called SigmaPlot™ was used as an environment in which to view these differences and a moderately difficult graphing task was constructed as the experimental task. The design of the study focused on the acquisition and incorporation of hot-key shortcuts into the user's strategy. It was assumed that the hot-keys were more efficient than mouse actions in that the former interaction technique typically combines a number of steps into a single keystroke. Therefore, if users do, in fact, gravitate toward the most efficient strategy as they gain experience, it would be expected that hot-keys would be used with greater frequency in subjects' later task trials. The data seem to suggest that experienced users do not necessarily adopt the most efficient strategy; instead the strategies fall into distinct categories, which are significantly correlated with performance on other cognitive tasks. A follow-on of the study is currently being conducted at George Mason University in conjunction with the Naval Research Laboratory.

The results of this study prompted several questions: Are all the hot-keys really more efficient? Are some more efficient than others and therefore have a greater "pay off" associated with their use? What interface design factors facilitate/hinder the acquisition and incorporation of hot-keys into the user's strategy? How does the computer interface design interact with differences in user strategies? In particular, are there identifiable patterns in user interface design that can be used to predict in which applications these differences are likely to occur? These questions suggest the need for an analytical specification method adequate to the purpose of representing sequence and interrelation among all possible user inputs in a given application's user interface during the

Manuscript approved March 1, 1995

performance of some task. A survey of existing formal specification methods was then conducted in an attempt to find such a method. Although the primary factors of interest were more likely to be addressed by techniques from the behavioral or user-centered domain, constructionalist or system-oriented methods were also considered.

2. SURVEY OF FORMAL INTERFACE SPECIFICATION TECHNIQUES

2.1. Introduction

The user interface is arguably one of the most critical components of any software designed for interactive use. A given user interface instantiates a set of methods and conventions by which a user and a computer software package interactively communicate to accomplish work. What makes the interface component so important is the issue of *usability*, that is, how easy or difficult a system is to learn and use. "The principal problem in building computer systems used to be providing sufficient processing power; now, it is more often providing a good user interface. However, the designer trying to engineer a good user interface is handicapped without a clear and precise technique for specifying such interfaces" (Jacob 1986). This section discusses different styles of interface interaction and the different approaches taken in specifying the interface. Formal and semi-formal techniques suitable for specifying the user interface of a computer system are surveyed and the relevant literature is reviewed.

2.1.1 Sequential vs Asynchronous Interaction

New styles of interaction (e.g., direct manipulation of graphical objects and icons) have more complex temporal behavior and are more difficult to represent than older styles of command languages and menus, which were largely constrained to predefined sequences (Hartson and Gray 1990). These new interaction techniques are asynchronous in nature with "many tasks available to the user at once, and sequencing within each task is independent of sequencing within other tasks" (Hix and Hartson 1993). In sequential interaction the control moves predictably through the dialogue and, hence, is easier to describe.

2.1.2 Constructional vs Behavioral Models

"Most representation techniques currently being used for interface software development (e.g., state transition diagrams, event-based mechanisms, object orientation) are constructional—and properly so. Any technique that can be thought of as describing interaction from the system viewpoint is constructional" (Hix and Hartson 1993). Behavioral representation techniques attempt to get away from software issues and into the issues that arise before software design, such as task analysis, functional analysis, task allocation, and user modeling. "Consequently, behavioral representation techniques and supporting tools tend to be user centered and task oriented" (Hix and Hartson 1993).

Although a few constructional interface design models are capable of capturing some temporal aspects of interface behavior, they are not capable of representing the temporal relationships within concurrent and asynchronous interaction (Green 1986). Conversely, most behavioral models completely fail to provide any method for specifying system actions. As each approach has its own view of the interface, neither should be considered sufficient, eliminating the need for the other. In fact, each supports a different domain, and if a full specification of an interface is desired, both approaches should be taken.

2.1.3 Qualities of a Good User Interface Specification

Jacob (1986) defines eight qualities that a specification technique for user-computer interfaces should possess. (1) *The specification should be easy to understand.* Specifically, it should be easier to understand, and possibly shorter, than the software that implements the interface. (2) *The specification should be precise.* It should explicitly specify the behavior of the system for each possible input. (3) *The specification technique should be powerful.* It should be able to express complicated system behavior across a wide variety of interfaces in a compact and easy to understand fashion. (4) *The structure of the specification should be closely related to the user's mental model of the system.* The specification should represent the cognitive structure of the user interface rather than the physical actions required or such implementation details as internal data or control structures. (5) *The specification should be easy to check for consistency.* It should make apparent inconsistencies and oversights in a user interface design and, in general, facilitate the evaluation of a user interface from its specification. (6) *The specification should emphasize the cognitive steps the user performs.* This emphasis is essential for using the specification to predict user performance. (7) *The specification should separate function from implementation.* It should describe the behavior of a user interface completely, precisely, and unambiguously but constrain the way in which it will be implemented as little as possible. (8) *Ideally a specification should be useful for simulating a user interface.* Given a specification, it should be possible to construct a prototype or mockup of the user interface of a system rapidly and, perhaps (with an executable specification), automatically.

2.2. Constructional Techniques

2.2.1. Backus-Naur Form (BNF)

Backus-Naur Form (BNF) is a technique based on production rule grammars and was originally designed to specify static languages (Backus et al. 1960). The language is described by a set of production rules, and from them, all possible valid inputs in the language can be produced. "Each rule gives the definition for some *nonterminal* symbol. Wherever that symbol itself appears (in the definition of some other symbol), it may be replaced by substituting the contents of its definition" (Jacob 1986). This allows the definition of a single starting symbol to ultimately yield all legal strings in the language. Although this method is capable of relating many design aspects of the user interface the specifications are often awkward and cryptic, and even indecipherable to the lay reader.

Several investigators have modified the standard static language BNF to specify interactive languages, such as user interfaces (e.g., Fenchel 1982; Lawson, Bertran, and Sanagustin 1978; Reisner 1981). Reisner's BNF specifies a nontrivial, real-world system but "describes only the syntax of the input language; it gives no idea of what (if anything) the system will do as the user is entering commands" (Jacob 1986). Jacob also notes that Reisner fails to overcome the traditional problems of BNF-based techniques—representation of interactive prompting, error handling, and correction that must occur at particular points in a dialogue. These are all awkward to specify, as controlling precisely when a grammar rule is satisfied frequently requires introducing many artificial intermediate constructs. In addition, the representation of optional "detours" (e.g., help functions) are not supported and temporal aspects of a user-interface are difficult to capture.

2.2.1.1 Multiparty Grammars

Shneiderman (1982) introduces a new type of BNF-based grammar called Multiparty Grammars. In this notation, each nonterminal symbol is associated with either user-input, the computer, or mixed in a multiway conversation (Jacob 1986). "User-input and computer

nonterminals represent user actions and computer responses, respectively. Mixed nonterminals represent sequences in the human-computer dialogs" (Carr 1994).

The Multiparty Grammars notation is successful in many ways; it manages to move both sides of the dialog into the syntactic domain, introduces a notation to represent visual attributes of display characters, and allows for the specification of multiple windows on a display. In its present form "Multiparty Grammars are good for modeling keyboard-based command language interactions but are very awkward for direct-manipulation interfaces" (Carr 1994).

2.2.2. State Transition Diagrams

"State Transition Diagrams have long been used as a graphical representation of interaction control flow in sequential interaction" (Hix and Hartson 1993) (also Wasserman 1985; Wasserman and Shewmake 1985; Jacob 1986). In its basic form, a transition diagram is a series of nodes and directed arcs. In this approach, the transition arcs represent state transitions based on user inputs, and the nodes represent interface states or screens (Carr 1994). Arcs are generally associated with actions, in which case "upon traversing the arc (executing the action), the terminal is then in the state represented by the node at the end of the arc" (Wasserman and Shewmake 1985). "Because the nodes of these diagrams usually do not represent interface feedback or screen appearance directly, the content of a node—a representation of what happens within that state—is often described in some other form, such as an *interface representation language*" (Hix and Hartson 1993). The same sort of problem arises with descriptions of actions. "The actions could be specified informally with a narrative, or more formally using a formal specification language, or assertions with preconditions and postconditions" (Wasserman and Shewmake 1985).

Despite their advantages, conventional state transition diagrams have several drawbacks which make them impractical for the specification of large, complex systems (Harel 1988; Wellner 1989). (1) Conventional state diagrams are "flat." They provide no depth, hierarchy or modularity, and therefore do not support top-down or bottom-up development. (2) Conventional state diagrams are uneconomical when it comes to transitions. An event that causes the same transition from a large number of states must be attached to each state separately. (3) Conventional state diagrams require exponential growth in states as the system grows linearly because every possible state must be explicitly represented. (4) Conventional state diagrams are inherently sequential and cannot easily represent concurrent activities.

Wasserman and Shewmake (1985) propose the User Software Engineering (USE) methodology, which extends the traditional state transition diagrams to include semantic content to provide for the executable specification of a system. The USE methodology also allows for the inclusion of variables into the notation, and it allows actions to "return a value, with branching to a node or subconversation dependent on that value" (Wasserman and Shewmake 1985). The transition arc concept was extended to: (1) handle buffered or unbuffered input, (2) handle extended keys on the terminal (e.g., function keys and arrow keys), (3) support the immediate recognition of particular keys (e.g., a function key or help key), (4) allow for the extension of the set of input string terminator keys, (5) apply a time limit on waiting for user input, after which a transition is made, (6) truncate user input to a fixed length, and (7) allow for a transition to be made with no input at all, just to perform an action. "Thus the USE methodology extends the concept of a transition diagram to handle situations that frequently arise in the construction of interactive information systems. As a result, USE transition diagrams are highly specialized to the specification of interactive systems and contain much more information than is available in classic state transition diagrams" (Wasserman and Shewmake 1985).

2.2.2.1 Concurrent State Diagrams

Jacob solved some of the combinatorial explosion problems of traditional state transition diagrams with the introduction of Concurrent State Diagrams (1986a). This method proposes "a mutually asynchronous set of state diagrams (that) represent the interface, offering the graphical advantage of a diagrammatic approach but avoiding the complexity of a single large diagram" (Hix and Hartson 1993). Multiple diagrams are active simultaneously, and control is transferred from one to the other in a coroutine fashion. "Coroutines do not solve the transition complexity problem for specifying some systems (notably those with context sensitive help)" (Carr 1994).

2.2.2.2 Executable Specification

Jacob (1983) describes a specification technique for user-computer interfaces that are executed interpretively to provide a working prototype of the system. The technique uses state transition diagrams to emphasize the time sequence aspects of the user interface and decomposes the specification into three components. Jacob adopts Foley and Van Dam's (1982) three levels representing "increasing amounts of specificity and detail and decreasing abstraction, from the semantic level to the lexical level" (Jacob 1985). The *semantic* level describes the functions performed by the system and tells what information is needed to perform each function and its result. The *syntactic* level describes the sequence of inputs and outputs. For inputs, this means describing the rules by which *tokens* in the language are formed into proper "sentences." The *lexical* level "determines how input and output tokens are actually formed from the primitive hardware operations (*lexemes*). It represents the binding of hardware actions to the hardware-independent tokens of the input and output languages" (Jacob 1985). The syntactic specification is then refined from an informal specification to a formal, executable one.

It is quite useful to be able to construct a prototype of the user interface directly from the specification. "While many prospective users will find a formal specification of a proposed system difficult to understand, they will have much less trouble evaluating a mockup system and identifying deficiencies in its user interface, both through informal demonstrations and formal experiments" (Jacob 1985). This technique has proven successful in specifying and constructing several prototypes built in the Secure Military Message Systems project at the Naval Research Laboratory.

2.2.3 Statecharts

Statecharts (Henderson 1986) are graphical notations that extend conventional state transition diagrams, attempting to preserve all of their benefits while overcoming their drawbacks (Wellner 1989). Harel has done much work on perfecting the use of statecharts as a specification method for user interface dialogues (Harel 1987; 1988) and has come up with a method which provides for a hierarchical representation that makes it easy to see at a glance how the interfaces are organized. The three fundamental components of these statecharts are *states*, *events*, and *actions* with behaviors being made up of Event-Action pairs (Wellner 1989).

Statecharts gave state diagrams the ability to represent two new kinds of states to represent the concepts of *grouping* and *concurrency*. If several states have identical transitions triggered by the same event, then those states can be grouped together in a single meta-state, and a single transition can be specified from the group instead of several. "Meta-states are divided into two types: parallel or AND-states and sequential or XOR-states. Meta-states enclosed within AND-states may execute in parallel and fulfill the function of coroutines" (Carr 1994), whereas only one state inside of a XOR meta-state can be active at one time.

Statechart specifications represent all interactions with the outside world through *events* and *actions*. Input is represented by events and output by actions. "Events can be generated by the I/O drivers, interaction technique objects, the application software, or by the *broadcast* action" (Wellner 1989). Any state can have entry and exit actions. (The entry actions are initiated whenever the state is entered, and the exit actions when it is exited.) *Broadcast* events are actions which generate an event that appears the same as an externally generated event and are used to specify a user interface where one part of the dialog effects another. Conditional actions check the value of conditional variables or the active status of specific states to determine which action to execute. These actions guard the entrance to specific states, imposing constraints on the dialog, and handle exceptions (Wellner 1989).

Statecharts are an effective specification technique for communicating dialog designs and modifications because they are easily understood, formal, and compact. The major drawback to Statecharts is "as originally defined [they] do not incorporate data flow or abstraction" (Carr 1994).

2.2.3.1 Interface Representation Graphs (IRGs)

Interface Representation Graphs (IRGs), introduced by Rouff in 1991, extend the statechart to represent dialog. IRGs maintain the traditional statechart form while introducing several new extensions. "IRG nodes represent a physical or logical component of an interface as well as a state" (Carr 1994). IRGs are capable of specifying data flow as well as control flow, and constraints are supported. IRGs can also represent inheritance of interface objects, data flow, control flow, and attributes. "Finally, to support UIMS (User Interface Management Systems) functionality, IRGs permit specification of semantic feedback between the application and the user interface" (Carr 1994). IRGs provide several useful extensions to the traditional statechart and are used successfully as the underlying representation technique in Rouff's Rapid Program Prototyper (RPP).

2.2.3.2 Interaction Object Graphs (IOG)

The Interaction Object Graph (IOG) is a new statechartlike method for specifying interaction objects or widgets developed by Carr (1994). The IOG draws from IRGs for its data flow and constraint specifications, UAN (discussed below) for its description language, and the traditional statechart for its transition diagram execution model. Not only does the IOG provide for a more condensed specification than traditional methods, it gives the reader a clear idea of both the object's appearance and the dynamic changes in its appearance. This technique has proven quite successful at clearly specifying simple interaction objects, but a complete specification of a complex widget becomes quickly "muddled." Current efforts are focused on designing a tool that will allow for the direct execution of IOGs.

2.2.4 Object-Oriented Approach

Object-oriented systems, such as those proposed by Sibert, Hurley, and Blesser (1988), fit today's interaction styles well as "the behavior of interaction objects is naturally event driven and asynchronous" (Hix and Hartson 1993). In such an environment, "the behavior of a certain element may be predefined as an attribute of that element's property list. Only when it is necessary to modify the default behavior of a particular object does it become necessary to explicitly specify a portion of the user interface logic" (Wilson and Rosenberg 1988). This provides for a much more compact specification than traditional approaches.

One of the major advantages of the object-oriented approach is that it provides a degree of autonomy from the internal software structures, which make up the interface (Sibert, Hurley, and

Blesser 1986). However, the object-oriented approach distributes the flow of control, which makes it difficult to understand or trace the sequencing (Hix and Hartson 1993).

2.2.5 Event Handlers

One type of constructional representation technique, similar to object-oriented programming and designed especially for asynchronous interaction, uses the concept of event handlers (e.g., Green 1985a). "With these, each user action or input is viewed by the system as an event and is sent to the appropriate *event handler*" (Hix and Hartson 1993). "An event handler is a procedure that performs a set of actions based on the name of the event it receives. These actions include passing output tokens to the presentation component, passing input tokens to the application interface model, performing some calculation, or generating new events" (Green 1985b). States are represented as the collection of events processed by an event handler, and "the set of event handlers active (able to receive events) at any one time defines the legal user actions at that point in the dialogue" (Green 1985). This model has proven quite effective and has been used in the Macintosh (Apple 1985; Chernicoff 1985) and in many highly interactive systems (Green 1985a; Hill 1986).

2.2.6 Multilayered Method

Foley and Van Dam's (1982) method is a multileveled model of interaction that allows both abstract and concrete details of interaction to be documented within the same scheme. The complete specification consists of four levels: the *conceptual design*, the *semantic design*, the *syntactic design*, and the *lexical design*. Each level is specified separately and has its own form.

"The conceptual design serves to outline the user's model of the application" (Foley and Van Dam 1982) and is said to comprise four things: (1) the set of all types of *objects* in the system, (2) the *relationships* between those types of objects, (3) the *properties* associated with the types of objects, and (4) the *operations* that can be performed on the objects, the relationships between objects, or the properties of objects. The semantic level of specification is concerned with the operations a user can perform, and each operation is defined in a table that describes the parameters of the operation, the system feedback, possible errors, and performance measures (Frohlich and Luff 1989). Foley, McCormick, and Bleser (1984) suggest that the syntactic level of the interface be specified using two notations, Finite State Diagrams and BNF notation. The lexical component is described in input lexicon tables and, for output lexicon, data specifications and message specifications.

This multilayered method is quite comprehensive and produces unusually long specifications. This method's specification of the semantic and syntactic levels of the interface is "clumsy" and "verbose and might be written more economically using a formal notation such as first-order logic" (Frohlich and Luff 1989). Despite these shortcomings, the method is quite effective and has potential to facilitate the design and development cycle.

2.2.7 Interaction by Demonstration: PERIDOT

PERIDOT is "a rule-based design tool in which the designer demonstrates by example how he wishes the interface to look and work" (Wilson and Rosenberg 1988). The system "learns" the common behaviors and applies them to similar situations within the interface, eliminating the need for describing all of the dialogue states. (Only the exceptions need be encoded.) PERIDOT introduces the virtual mouse, which allows for the direct specification of the behavior the designer wants the mouse to adopt when the interface is running. This approach, sometimes referred to as visual programming, significantly expedites the creation of direct manipulation interfaces.

"This is a novel and creative approach and is very suitable for producing rapid prototypes. However, it produces only program code, with no behavioral representation of the interface that can be analyzed" (Hix and Hartson 1993). Myers (1987) notes that "the designer's actions when manipulating the tool can be ambiguous. This technique is also not well suited to describing highly syntactic structures, such as how a textual input string should be parsed." "While specifying interface behavior by example is not suited to all forms of user dialogue, it can enhance the dialogue design process significantly, particularly when used in conjunction with other tools, such as a formal grammar mechanism (Wilson and Rosenberg 1988).

2.2.8 User Interface Development Environment (UIDE)

The User Interface Development Environment (UIDE), introduced by Foley, Gibbs, Kim, and Kovacevic (1988), allows alternative interfaces with the same basic functionality to be generated. Designing an interface in UIDE involves "building a *knowledge base* consisting of objects, attributes, actions, and pre- and post-conditions on actions" (Hix and Hartson 1993) that encode dynamic behavior and are used to describe partial semantics of application actions. "These partial semantics are used for many purposes, including selective enabling of menu items, (providing) partial explanations of what an action does, providing context sensitive animated help, applying correctness-preserving transformations to the interface, checking the completeness and consistency of the interface, and dialogue sequencing" (Gieskens and Foley 1992). In 1992, Gieskens and Foley extended the UIDE mechanism to include all interface objects, thus providing a much finer grain of control over the development of the user-interface.

2.2.9 Graphical Control Specification Systems: Panther

A graphical interface specification tool for UNIX®-based applications, called Panther, has been designed by Helfman (1987). "Unlike similar systems, which focus on combining interaction techniques, Panther allows the specification of low-level interactions by invoking user-selectable subroutines for input-device transitions" (Helfman 1987). It is flexible enough to allow both experts and novices to create, test, and modify configurations for application interfaces.

Panther contains its own window management system and, therefore, can run with no external window support. A Panther *interface* is made up of hierarchically nested rectangular *regions*. Each region can be used as a control device (e.g., buttons, knobs, and slider-bars), or it may be used to display and manipulate application-related data. Each region is defined in terms of seven attributes: name, coordinates, highlight style, draw flag (used to identify application-specific data that must be redrawn frequently to ensure the accuracy of displayed data), parent name (identifies the region's parent), draw routine, and selection routine (subroutines called when the region is selected). These attributes are specified in a tabular notation, which is then stored as an ASCII file until compiled.

"Panther was initially used to edit objects in a keyframe animation system" (Weil and Helfman 1984) and "has since been used to specify the interfaces for several paint systems and an image processing system" (Helfman 1987). The current Panther system uses textual tables as input; however, certain region attributes such as coordinates would be easier to manipulate in a graphical user interface that could change dynamically as it is specified. Helfman acknowledges this point and mentions that research on a Graphical User Interface (GUI) Panther is underway.

2.3. Behavioral Techniques

2.3.1 Goals Operators Methods and Selection Rules (GOMS)

"The GOMS model concept, originally introduced by Card, Moran, and Newell (1983), is one of the most widely accepted analytical modeling concepts in the HCI community" (Gong and Kieras 1994). Recent efforts to refine the GOMS model have led to a notation system NGOMSL (Natural GOMS Language), which allows "GOMS models to be written down with a high degree of precision, but without the syntactic burden of ordinary formal languages and that is also easy to read rather than cryptic and abbreviated" (Kieras 1988). Unlike traditional task analyses, which focus on the construction of an action/object table, a GOMS task analysis describes in detail the specific method for accomplishing the goals listed in the action/object table. Thus, a GOMS task analysis begins where traditional task analysis leaves off (Kieras 1988).

The GOMS model describes the user's cognitive structure in terms of four components: "(1) a set of **G**oals, (2) a set of **O**perators, (3) a set of **M**ethods for achieving the goals, and (4) a set of **S**election rules for choosing among competing methods for goals" (Card et al. 1983). A goal is something the user tries to accomplish, and a goal description is an action-object pair in the form <verb-noun>. Goal lists are typically represented hierarchically with the accomplishment of a goal usually requiring the completion of one or more subgoals. "Operators are elementary perceptual, motor, or cognitive acts, whose execution is necessary to change any aspect of the user's mental state or to affect the task environment" (Card et al. 1983). Operators, like goals, take the action-object form, but a goal is something to be accomplished whereas an operator is simply executed. Methods are sequences of goals and operators that describe a procedure for accomplishing a goal. Selection rules are a set of if-then notations that are used to route control to the appropriate method to accomplish the goal.

While the GOMS method currently only models error-free performance, Card et al. (1983) suggest that the notation might eventually be extended to cover errors. When the Methods, Operators, and Selection Rules are very large in number or unidentifiable, it is difficult to predict behavior from the specification (Koubek et al. 1989). Nevertheless, it is quite effective in identifying usability bottlenecks, providing guidance for design solutions, and producing useful quantitative predictions for design alternatives. Recent studies have documented the model's validity and have demonstrated that "the GOMS model method can and should have a role in the iterative cycle of software interface design and evaluation" (Gong and Kieras 1994), (Nielsen and Phillips 1993).

2.3.2 Keystroke-Level Model

The Keystroke-Level Model (Card, Moran, and Newell 1980) is a Keystroke Level GOMS analysis refined into a model of practical use. It is a system design tool intended for the purpose of predicting one aspect of performance—the time it takes an expert user to perform a given task on a given computer system. Although quite powerful, the Keystroke-Level Model has several restrictions: "The user must be an expert; the task must be a routine unit task; the method must be specified in detail; and the performance must be error-free" (Card et al. 1980). The authors note that these factors severely limit the model's application but insist that this level of specificity is necessary to get a valid model that accurately predicts the time taken to perform a task.

The method is simple. The interface is broken down into a series of small, cognitively manageable, quasi-independent tasks that are referred to as unit tasks. The unit tasks are considered to consist of two parts: the acquisition of the task and the execution of the task. The total time to perform the given task is the sum of these two parts. "The Keystroke-Level Model asserts that the execution part of a task can be described in terms of four different physical-motor operators: **K** (keystroking), **P** (pointing), **H** (homing), and **D** (drawing), and one mental operator, **M**, by the user,

plus a response operator, **R**, by the system" (Card et al. 1980). The task is written in terms of these operators, each of which has a time value associated with it. "The evaluator then counts the number of occurrences of each type of operation that appear, multiplies each total by [its] time constant, and adds the components together" (Roberts 1988). This is then summed with the acquisition constant to give the total time taken to complete the given task.

The Keystroke-Level Model, despite its limitations, has proven to be a useful tool to developers. "A number of empirical studies have shown that the predictions of GOMS and the Keystroke Model are reasonably accurate, and that sometimes one can even use the same time parameters across applications" (Carroll and Olson 1988). Card, Moran, and Newell (1983) demonstrated the consistency of their parameters across text processors, operating systems, and graphics packages. An experiment by Olson and Nilsen (in press) showed that the basic parameters applied to spreadsheet software as well.

2.3.3 Command Language Grammar (CLG)

Moran's Command Language Grammar (CLG) takes a top-down approach and uses a LISP-like notation in a representational scheme designed to "describe the user's conceptual model of the system" (1981). The CLG divides the system into components and levels, from an overall task analysis to individual key presses. The three components Moran describes are a Conceptual Component (abstract concepts and tasks), a Communication Component (command language and conversational dialogue), and a Physical Component (the physical devices that the user sees and comes in contact with).

Each of these components comprises two levels, with each level being a complete description of the system at its level of abstraction and a refinement of the previous levels. The Conceptual Component is made up of the *task* level, which describes the task domain addressed by the system, and the *semantic* level, which describes the concepts represented by the system. The Communication Component consists of the *syntactic* level, which describes the command-argument structure, and the *interaction* level, which describes the dialogue structure. The Physical Component contains the *spatial layout* level, which describes the arrangement of the input/output devices and the display graphics, and the *device* level, which describes all the rest of the physical features.

The CLG seems to "presuppose a complete and explicit design before it can generate any representation [and therefore] cannot support the design process, but rather must be supported by it" (Carroll and Rosson 1985). Carroll and Rosson also point out that the method fails to take into account user errors and, therefore, does not consider them a relevant factor in user-interface design or in the psychological model of the user. Jacob notes that this multilayered format "results in an unusually long and detailed specification" (1986b). In a study conducted by Sharatt (1987), postgraduate students were asked to design and specify a relatively simple user-interface, using Moran's CLG. The students had difficulty interpreting the methods described by Moran, and only succeeded in generating lengthy specifications that required enormous amounts of effort. The specifications themselves proved incomplete, lacking adequate descriptions of the physical components, and there was no evidence suggesting that the evaluation metrics applied to the CLG specifications had any psychological validity (Frohilich and Luff 1989).

2.3.4 Task-Action Grammar (TAG)

Task-Action Grammar, introduced by Payne and Green (1986) as a formal model of the mental representation of task languages, attempts to model the user's knowledge by rewriting tasks into action specifications. The semantic structure of the command language is then mapped onto syntactic structure, thereby allowing the reader to make predictions on the learnability and usability

of the interface. "The basic principle of TAG is that, through the apparatus of semantic features, task tokens can denote well-defined categories of tasks" (Payne and Green 1986).

While a TAG description's format of breaking the interface into Simple Tasks, Task Features, and Rule Schemas provides for a clear description of the individual tasks and highlights the consistencies, it also produces quite a lengthy specification. Another of TAG's weaknesses is that its "simple feature-based semantics cannot cope with all the intricacies of users' conceptual models, so that important HCI considerations that are within the broad purview of the model (such as the psychological impact of lexical names, or the perceived consistency of object oriented interfaces) remain outside its scope" (Payne and Green 1989).

2.3.5 User Action Notation (UAN)

User Action Notation (UAN) is a behavioral representation technique developed at Virginia Tech by Hartson, Siochi, and Hix (1990). UAN focuses on both the user and the interface, describing the interaction between the two while performing a specified task. Although many constructionalist approaches allow for specification at multiple levels of abstraction, among behavioral approaches, UAN is the only representation technique surveyed that does not force a detailed low-level description; rather it allows one to choose the desired level of abstraction. Another strength of the UAN as a task description language is it's provision for specification of some aspects of system behavior. However, as Carr points out, it still "concentrates heavily on describing user actions and is not well adapted to describing software state" (1994).

The UAN's format is tabular in nature and divides the interface into asynchronous tasks that are quasi-hierarchically related to one another (Hix and Hartson 1993). The notation is read from top to bottom and left to right with sequence represented by rank, from top to bottom, and lateral alignment representing associated behavior. As originally described, the notation is a table divided into four columns: User Actions, Interface Feedback, Interface State, and Connection to Computation. Hix and Hartson (1993) suggest that a UAN task description be complemented with screen pictures and scenarios, interface state transition diagrams, and design rationale descriptions.

The UAN's major contributions are its compact—yet powerful—description language and its ability to clearly and simply express large and complex interface designs at multiple levels of abstraction. "The UAN, at present, is awkward at specifying system responses to unexpected user actions and, due to its tabular format, does not clearly represent relationships between tasks" (Carr 1994). Nevertheless, "the User Action Notation provides a crucial articulation between the behavioral domain and design and implementation in the constructional domain" (Hartson and Gray 1990).

3. THE USER ACTION NOTATION

None of the above surveyed techniques was entirely capable of capturing and clearly representing interfaces with parallel path structure and showing the comparative efficiency of functionally equivalent paths. Nevertheless, the UAN, because of its versatility, demonstrated the most strength in this area and, furthermore, showed strong potential for use as an analytical specification method. The remainder of this paper focuses on the authors' modification and analytical use of the UAN for specifying interfaces with multiple input devices and parallel path structure.

As previously mentioned, UAN focuses on both the user and the interface, describing the interaction between the two while performing a specified task. This provides an ideal environment in which to study the interaction between user strategy and the computer interface design. The specification's tabular format clearly represents the interaction between the user and the interface

through its representation of associated behavior by lateral alignment across columns. The clear representation of associated behavior is necessary to determine functional equivalence and the relative efficiency of the functionally equivalent methods. As previously mentioned, the notation is read from top to bottom and left to right with sequence represented by rank, from top to bottom. At the articulatory level the table is divided into four columns: User Actions, Interface Feedback, Interface State, and Connection to Computation (see Fig. 1).

Task: **save file**			
USER ACTIONS	**INTERFACE FEEDBACK**	**INTERFACE STATE**	**CONNECTION TO COMPUTATION**
((~[file menu] @menu-bar Mv	∀ menu label-! file menu label! display (file menu)	selected= file menu	file functions selected
M ^) OR (R: Alt-F v^)	new menu choice! *(elicits all feedback in this cell)*	default= new function *(accomplishes all interface state changes in this cell)*	*(elicits all computations in this cell)*
((~[save]@file menu Mv	save menu choice!	selected= ∅ selected= save function	
M^)	erase (file menu) display (saving data message box) saving finished: erase (saving data message box)	selected= ∅	updated file' saved
OR ((↑↓)* > [save menu choice])	save menu choice!	selected= ∅ selected= save function	
Enter v^) OR (S v^))	erase (file menu) display (saving data message box) saving finished: erase (saving data message box)	selected= ∅	updated file' saved
OR (R: Ctrl-S v^)	*(elicits all distinct feedback in both cells of this column)*	*(accomplishes all distinct interface state changes in both cells of this column)*	*(accomplishes all distinct computations in both cells of this column)*

Fig. 1—UAN specification of the SigmaPlot **save file** task

Figure 1 is a UAN description of the SigmaPlot™ **save file** task. The first cell of the User Actions column in this specification is read move (~) to the file menu at the menu-bar ([file menu]@menu-bar), press the mouse button down (Mv), then release the mouse button (M^); or (OR) recall (R:) and depress the Alt and F keys simultaneously (R: Alt-F). The parentheses in the notation serve as grouping mechanisms for sub-tasks. In this specification the action of moving to the file menu at the menu-bar is associated with the Interface Feedback of all menu labels becoming unhighlighted (menu label-!), the file menu label becoming highlighted (file menu label!), and the file menu being displayed (display (file menu)); the Interface State of the file menu being selected (selected= file menu); and the Connection to Computation of file functions being selected (file functions selected). Similarly, the action of releasing the mouse button is associated with the interface actions which are laterally aligned with the notation (M^). It is understood that if the Alt-F method is used, then it is associated with all the interface actions in that row. For a complete defined list of UAN symbols used in this specification, see Appendix A.

4. MODIFICATIONS TO THE UAN

In general, the UAN's ability to represent user actions in a clear and concise format makes it particularly useful as an iterative design and analysis tool. In what follows, modifications to the format are proposed with the aim of increasing its analytical strengths and enhancing the method's ability to specify complex interfaces with parallel path structure.

While the notation does provide for a clear description of the user's actions and a connection to the corresponding system behavior when one input device is being used (such as a mouse), the structure becomes confusing when, as in the previous example, there are many ways of completing the same task and more than one input device is available (e.g., a keyboard and a mouse). The problem lies in the representation of sequence and association. In the previous example, there are multiple ways of completing specific actions, and this is represented by employing the symbol OR. However, in order to maintain the lateral association and top to bottom sequencing, choice must be represented at each step of the specified task. Figure 1 illustrates how confusing the specification of a relatively simple task can become. The User Action specifications are difficult to follow, and it is difficult to maintain the representation of associated behavior by lateral alignment. It becomes necessary to maintain lateral association by repeating many of the system actions. This method yields an unduly long and confusing specification. In fact, to provide a truly accurate specification, all of the system responses should be repeated *again* aligned with the **(**R: Ctrl-S v^**)** notation. In complex tasks containing many steps, this method becomes even more confusing to the reader of the specification. Since one of UAN's major aims is task specification in a clear, easily understood format (Hartson and Gray 1990), this cluttering of the specification is undesirable.

4.1 Dividing the User Actions Column

When a task within an application can be performed in more than one way (i.e., with more than one input device), the specification can be made much more clear if the User Action column is divided into separate columns, one for each input device (e.g., mouse and keyboard). If the User Actions column in Fig. 1 is divided to accommodate separate input devices, then the notation allows for concurrent specification of equivalent mouse and keyboard functions and becomes as follows (see Fig. 2):

TASK: **save file**				
USER ACTIONS		**INTERFACE FEEDBACK**	**INTERFACE STATE**	**CONNECTION TO COMPUTATION**
mouse	**keyboard**			
~[*file menu*]@menu-barMv	(R: Alt-F v^)	∀ menu label-! *file* menu label! display(*file* menu)	selected= *file* menu	*file* functions selected
M^		new menu choice!	default= *new* function	
~[*save*]@*file* menu Mv	((↑↓)* > [*save* menu choice]	*save* menu choice!	selected= ∅ selected= *save* function	
M^	Enter v^) OR (S v^)) OR	erase (*file* menu) display (*saving data* message box) saving finished: erase (*saving data* message box)	selected= ∅	updated *file*' saved
	(R: Ctrl-S v^)	*(elicits all feedback in both cells of this column)*	*(accomplishes all interface state changes in both cells of this column)*	*(accomplishes all computations in both cells of this column)*

Fig. 2—UAN specification of the SigmaPlot **save file** task with User Actions column divided to accommodate separate input devices

Notice that this specification maintains lateral alignment's representation of associated behavior, even between input devices, and eliminates the need for excessive use of the UAN symbol OR, making the specification much more clear and concise (with the exception of the (R: Ctrl-S) hot-key specification, which is discussed later). The User Actions column is separated from the Interface Actions columns by a double line, and the italicized Times font is used to denote terms that are specific to the specified application and not to UAN (i.e., menu names, and dialogue box names). These cosmetic changes are introduced solely for clarification. The table is still read from top to bottom and left to right with association still represented by being on the same line. However, within the User Actions column, the state of being on the same line represents functional equivalence. Therefore, there exists an implied OR at the meeting of adjacent mouse and keyboard cells in the User Actions column. Furthermore, every intersection of bold lines within the User Actions column represents a point at which the user may shift input devices while keeping on the task, as seen in Fig. 3. With this division, the User Actions column becomes a matrix that provides a clear and concise way of representing all the possible combinations of actions the user may employ to complete the specified task. For example, the user may use all mouse commands, all key commands, or any combination of the two.

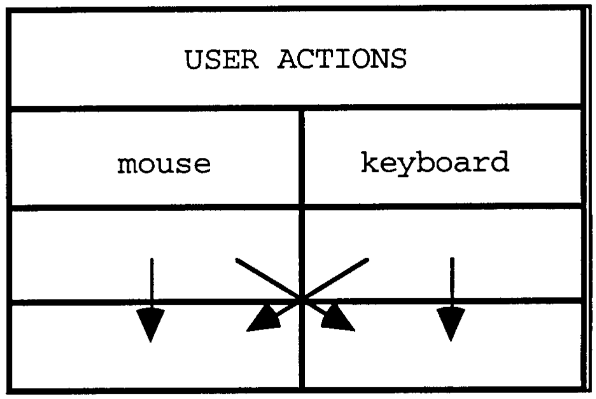

Fig. 3—Points at which the user may switch input devices

4.2 Hot-Key Specification

In some cases, there may be more than one way to complete a task using a particular input device. In these situations, it is possible to further divide the specific input device column (e.g., mouse or keyboard) to accommodate these alternative methods. These divisions represent an implied OR and are made by a thin line as they do not require a shift in input devices. Alternatives such as these are typically hot-keys or other kinds of shortcuts. This method of specification makes the benefit afforded by particular shortcut stand in relief.

Figure 4 schematically illustrates just such a situation. In this case, after completing action 1 or 1', the user may execute two mouse actions (2 and 3), two keyboard commands (2' and 3'), or use a hot key (2"), which will accomplish both actions, maintain the same interface response, and move the user to the same place within the application (4/4'). Note that the previously described ability of the user to shift from one input device to the other at the intersections applies to both boxes in the keyboard column. These actions, which cross cell boundaries, can only be initiated at the cell from which they originate. The user may proceed from 1 to 2; shift from 1 to 2'; or use the specified hot-key and shift from 1 to 2". The user who chose the hot-key command would then proceed to either 4 or 4' where the non-hot-key user would have to complete the action in either the cell denoted by 3 or 3'. This particular shortcut allows the user to execute two commands with one action.

<table>
<tr><td colspan="6">TASK:</td></tr>
<tr><td colspan="3">USER ACTIONS</td><td rowspan="2">INTERFACE FEEDBACK</td><td rowspan="2">INTERFACE STATE</td><td rowspan="2">CONNECTION TO COMPUTATION</td></tr>
<tr><td>mouse</td><td colspan="2">keyboard</td></tr>
<tr><td>1</td><td colspan="2">1'</td><td></td><td></td><td></td></tr>
<tr><td>2</td><td>2'</td><td rowspan="2">2''</td><td></td><td></td><td></td></tr>
<tr><td>3</td><td>3'</td><td></td><td></td><td></td></tr>
<tr><td>4</td><td colspan="2">4'</td><td></td><td></td><td></td></tr>
</table>

Fig. 4—Modified UAN format for specifying hot-keys

The SigmaPlot **save file** task illustrates a case where a second division within the keyboard column would be warranted. Once this modification is applied to the specification, it serves to further clarify the description. The new specification (Fig. 5) now illustrates the fact that employing the Ctrl-S shortcut is functionally equivalent to clicking on the file menu label and then on the save label in that menu. The Ctrl-S action is also equivalent to depressing the Alt and F keys simultaneously to display the File menu then selecting the Save function by depressing the S key or by navigating to the save label with the arrow keys and depressing Enter.

TASK: **save file**					
USER ACTIONS			**INTERFACE FEEDBACK**	**INTERFACE STATE**	**CONNECTION TO COMPUTATION**
mouse	**keyboard**				
~[*file* menu]@menu-barMv M^	R: Alt-F v^	R: Ctrl-S v^	∀ menu label-! *file* menu label! display(*file* menu) *new* menu choice!	selected= *file* menu default= *new* function	*file* functions selected
~[*save*]@*file* menu Mv	((↑↓)* > [*save* menu choice]		*save* menu choice!	selected= ∅ selected= *save* function	
M^	Enter v^) OR (S v^)		erase (*file* menu) display (*saving data* message box) saving finished: erase (*saving data* message box)	selected= ∅	updated *file*' saved

Fig. 5—UAN specification of the SigmaPlot **save file** task with the hot-key modification

4.3 Individual Input Actions

While this specification clearly represents all seven ways of completing the **save file** task, the specification can yield even more information if each cell in the User Actions column is divided into individual input actions, which we define as user actions that elicit a system response. These divisions are marked by thin lines and should not be confused with the thick divisions that divide subtasks and input devices. By dividing the cells this way, each division can be considered relatively equivalent, and the specification then yields information as to the relative efficiency in terms of user input actions required by the different methods to complete the specified task.

In Fig. 6, the specification has been divided into individual input actions. The action of moving to the file menu and depressing the mouse is equivalent to the action of releasing the mouse button, depressing the Alt and F keys, depressing the Enter key, etc. Moving the mouse cursor to the object of interest is not considered an individual input action, as there is no interface response until the button is depressed. (It is possible to break the cells down to individual *user* actions, in which case the act of moving the mouse cursor would warrant a division, and a similar representation would be

necessary in the keyboard column to represent the user's hands moving to the key(s) of interest.) Navigating a menu or dialogue box with arrow keys or the Tab key is considered one action, as the user need only locate the key(s) once and then it(they) can be pressed rapidly and used as a navigating device. In some cases, the user may be required to type in a character string (i.e., a file name). In these cases the input action of typing the string may have to be weighted more heavily than the other actions in the specification.

TASK: **save file**						
USER ACTIONS				**INTERFACE FEEDBACK**	**INTERFACE STATE**	**CONNECTION TO COMPUTATION**
mouse	**keyboard**					
~[*file* menu]@menu-barMv	R: Alt-F v^			∀ menu label-! *file* menu label! display(*file* menu)	selected= *file* menu	*file* functions selected
M^				*new* menu choice!	default= *new* function	
~[*save*]@*file* menu Mv	(↑↓)* > [*save* menu choice]	S	R: Ctrl-S v^	*save* menu choice!	selected= ∅ selected= *save* function	
M^	Enter v^			erase (*file* menu) display (*saving data* message box) saving finished: erase (*saving data* message box)	selected= ∅	updated *file*' saved

Fig. 6—UAN specification of the SigmaPlot **save file** task with individual input actions divisions

This modification makes all the possible paths through the task stand in relief and allows the reader of the specification to evaluate the different methods of completing a task in terms of how many individual input actions are necessary for its completion. In the previous example, the most efficient method of completing the **save file** task is the one input action of depressing the Ctrl and S keys simultaneously. The all-mouse method requires four input actions and is, therefore, arguably less efficient.

When making these types of comparisons, there are several factors that must be considered. In the **save file** task, the Ctrl-S method is the most efficient in terms of input actions, assuming that the user recalls the command. If not, the method may actually be less efficient, as the user may need to look up the command or try several different commands before finding the correct one. We have also already noted that mouse movement has a cost but is not an input action by this paper's definition. Another consideration to take into account when evaluating the efficiency of different methods is the cost associated with shifting between input devices. Take, for example, the following scenario: the user moves the mouse to the file menu and "clicks" on it, then switches to the keyboard

and depresses the S key. Here the "cost" of the shift could be considered to be the equivalent of another input action. Thus the sequence described in the this scenario is equivalent to four "input" actions—one for moving to the *file* menu and depressing the mouse button, one for releasing the mouse button, one for switching input devices, and one for pressing the S key. This description more accurately reflects the true cost of the user's path of actions. Nevertheless, there are also many simple keyboard commands that the practiced user might enact with a minimum of cost in movement, expending only the cost of a shift in focus.

These modifications not only make the UAN specification of interfaces with multiple input devices much more clear, they allow the reader of the specification to make meaningful comparisons of alternative methods for completing a specified task. For a complete specification of the **`SigmaPlot™ graphing task`** in the modified UAN format, see Appendix B.

4.4 Additions to the UAN Repertoire

Aside from these structural modifications, a few new symbols to implement the UAN repertoire follow:

↑↓,← →, Tab	These symbols are used in the User Actions column to specify key commands used to navigate dialogue boxes and menus. They are quite frequently used in this context: `(OR(↑↓,← →, Tab)* > [menu'])`, which reads as use the up, down, left, right arrows or the Tab keys zero or more times to move to the specified menu.
>	This symbol is defined as "to" (as in the previous example).
@	This symbol specifies location
R	The boldfaced R is used to denote recall of an obscure command by the user, such as the Ctrl-F hot-keys. Such commands are referred to as obscure, as they have no cues associated with them. (There is no cue to the user that Ctrl-F1 initiates the select graph function.)
R	The nonboldface R is used to refer to the recall of a key command for which the user is prompted (i.e., an intuitive command such as using Alt-F to access the File menu).
default=	This notation is used in the Interface State column to denote the default value/field of the specified menu, or dialogue box.
italicized Times	The italicized Times font (as opposed to the UAN-standard courier font) is used for terms specific to the application and not UAN, such as menu names, dialogue box names, and function names.
	The shaded cell represents an empty cell. This is used in cases where there is no equivalent to the action represented in the adjacent cell and, therefore, the user must complete the action in the unshaded cell.

5. DISCUSSION

The modifications proposed in this paper eliminate the need for multiple `OR` notations and grouping mechanisms necessary for representing optionality thereby maintaining the representation

The modifications proposed in this paper eliminate the need for multiple OR notations and grouping mechanisms necessary for representing optionality thereby maintaining the representation of associated behavior through lateral alignment and reducing the "clutter" of the User Actions column. The concept of functional equivalence between input actions and between subtasks is introduced and clearly represented by adjacency within the User Actions column. The matrix design of the User Actions column illustrates exactly where in the task the user may shift input devices while keeping on task by using intersecting bold lines to divide the cells and the input device columns. Finally, by dividing the User Actions into individual input actions, the notation yields a clear representation of all the possible paths through the task and allows the analyst to compare these methods in terms of the number of these actions necessary for completion.

The issues of comparative efficiency of user strategies brought to light by the modifications to the UAN presented here suggest that it would be possible next to design and write a program that will take the modified UAN specification as input and, from that, generate a list of all possible paths through the task. Furthermore, predictions of path completion times could be computed. This could be achieved by mapping each component action to a performance time value. As an example, times for keyboard and mouse actions could be determined, respectively, by using the values suggested by the Keystroke Level Model of human performance and Fitt's Law as applied to mouse movement (Card, Moran, and Newell 1984). Time values would also need to be included for any cognitive operators used in the specification as well as the system response times. The program could also be designed to accept human data on the specified task as input and, from that, compute frequencies associated with each path. This sort of information would clearly be useful for HCI research purposes such as identifying interface inconsistencies and analyzing data collected in usability studies.

6. ACKNOWLEDGMENTS

The authors thank Ben Shneiderman for sponsoring this work at the University of Maryland at College Park; Tucker Maney for the use of the PC without which this work would not have been possible.; and Astrid Schmidt-Nielsen, Debbie Hix, Bill Lawless, Dave Carr, Lisa Achille, Manuel Perez, and Helen Gigley for their insightful comments and suggestions; The User Action Notation was created and formalized at Virginia Tech by Hartson, Siochi, and Hix.

REFERENCES

Apple Computer, *Inside Macintosh* (Addison-Wesley, Reading, MA, 1985).

J. W. Backus, et al., "Report on the Algorithmic Language ALGOL 60," *Communications of the ACM* **3**, 299-314 (1960).

S. K. Card, T. P. Moran, and A. Newell, "The Keystroke Level Model for User Performance Time with Interactive Systems," *Communications of the ACM* **23**, 396-410 (1980).

S. K. Card, T. P. Moran, and A. Newell, *The Psychology of Human-Computer Interaction* (Lawrence Erlbaum Associates, Hillsdale, NJ, 1983).

D. Carr, "Specification of Interaction Objects," Proceedings of the ACM CHI'94 Celebrating Independence 1994, pp. 372-378.

J. M. Carroll and M. B. Rosson, "Usability Specifications as a Tool in Iterative Development," in *Advances in Human Computer Interaction*, H. R. Hartson, ed. (Ablex Publishing, Norwood, NJ, 1985) pp. 1-28.

S. Chernicoff, *Macintosh Revealed* (Hayden Book Company, Hasbrouck Heights, NJ, 1985).

R. S. Fenchel, "An Integral Approach to User Assistance," *ACM SIGSOC Bulletin* **13**, 98-104 (1982).

J. Foley, C. Gibbs, W. Kim, and S. Kovacevic, "A Knowledge-Based User Interface Management System," Proceedings of CHI Conference on Human Factors in Computing Systems, New York: ACM, 1988, pp. 67-72.

J. Foley, K. McCormick, and T. Bleser, "Documenting the Design of User-Computer Interfaces," Technical Report, Computer Graphics Consultants, Inc., Washington, DC (1984).

J. Foley and A. Van Dam, *Fundamentals of Interactive Computer Graphics* (Prentice-Hall, Englewood Cliffs, NJ, 1982).

D. M. Frohlich and P. Luff, "Some Lessons from an Exercise in Specification," *Human Computer Interaction* **4**, 121-147 (1989).

D. F. Gieskens and J. D. Foley, "Controlling User Interfaces Objects Through Pre- and Post-Conditions," Proceedings of ACM CHI'92 Conference on Human Factors in Computing Systems, 1992, pp. 189-194.

R. Gong and D. Kieras, "A Validation of the GOMS Model Methodology in the Development of a Specialized, Commercial Software Application," Proceedings of the ACM CHI'94 Celebrating Independence, 1994, pp. 351-357.

M. Green, "The University of Alberta User Interface Management System," *Computer Graphics* **19**(3), 205-213 (1985a).

M. Green, "Report on Dialogue Specification Tools," in *User Interface Management Systems*, G. E. Pfaff ed. (Springer-Verlag, Berlin, 1985b) pp. 10-14.

M. Green, "Design Notations and User Interface Management Systems," in *User Interface Management Systems*, G. E. Pfaff ed. (Springer-Verlag, Berlin, 1985c) pp. 89-107.

M. Green, "A Survey of Three Dialogue Models," *ACM Transactions on Graphics* **5**(3), 244-275 (1986).

D. Harel, "Statecharts: A Visual Formalism for Complex Systems," *Scientific Computer Programming* **8**(3), 231-274 (1987).

D. Harel, "On Visual Formalisms," *Communications of the ACM* **31** (5), 514-530 (1988).

H. R. Hartson and P. Gray, "Temporal Aspects of Tasks in the User Action Notation," *Human Computer Interaction* **7**, 1-45 (1990).

H. R. Hartson, A. C. Siochi, and D. Hix, "The UAN: A User-Oriented Representation for Direct Manipulation User Interfaces," *ACM Transactions on Information Systems* **8**(3), 181-203 (1990).

J. I. Helfman, "Panther: A Specification System for Graphical Controls," Proceedings of ACM CHI+GI '87 Conference on Human Factors in Computing Systems and Graphics Interface, 1987, pp. 279-284.

R. D. Hill, "Supporting Concurrency, Communication, and Synchronization in Human-Computer Interaction—The Sassafras UIMS," *ACM Transactions on Graphics* **5**, 179-210 (1986).

D. Hix and H. R. Hartson, *Developing User Interfaces: Ensuring Usability Through Product and Process* (John Wiley and Sons, New York, 1993).

R. J. K. Jacob, "Executable Specifications for a Human-Computer Interface," Proceedings of CHI'83, 1983, pp. 28-34.

R. J. K. Jacob, "An Executable Specification Technique for Describing Human-Computer Interaction," in *Advances in Human Computer Interaction,* H. R. Hartson, ed. (Ablex, Norwood, NJ, 1985), pp. 211-242.

R. J. K. Jacob, "A Specification Language for Direct-Manipulation User Interfaces," *ACM Transactions on Graphics* **5**(4), 283-317 (1986a).

R. J. K. Jacob, "Survey and Examples of Specification Techniques for User-Computer Interfaces," NRL Report 8948, April 1986b.

D. E. Kieras, "Towards a Practical GOMS Model Methodology for User Interface Design," in *Handbook of Human-Computer Interaction*, M. Helander, ed. (Elsevier Science Publishers B. V., Amsterdam, North-Holland, 1988).

R. J. Koubek, G. Salvendy, H. E. Dunsmore, and W. K. LeBold, "Cognitive Issues in the Process of Software Development: Review and Reappraisal," *International Journal of Man-Machine Studies* **30**, 171-191 (1989).

H. W. Lawson Jr., M. Bertran,and J. Sanagustin, "The Formal Definition of Man/Machine Communication," *Software—Practice and Experience* **8**, 51-58 (1978).

T. Moran, "The Command Language Grammar: A Representation for the User Interface of Interactive Computer Systems," *International Journal of Man-Machine Studies* **15**, 3-50 (1981).

B. A. Myers, "Creating Dynamic Interaction Techniques by Demonstration," Proceedings of CHI'87, 1987, pp. 271-278.

B. A. Myer and W. Buxton, "Creating Highly Interactive and Graphical User Interfaces by Demonstration," Proceedings of the ACM SIGGRAPH '86, 1986, pp. 249-258.

J. Nielsen and V. L. Phillips, "Estimating the Relative Usability of Two Interfaces: Heuristic, Formal and Empirical Methods Compared," Proceedings of INTERCHI '93, New York: ACM, 1993, pp. 214-221.

J. Reitman Olson and E. Nilsen, "Cognitive Analysis of People's Use of Spreadsheet Software," *Human-Computer Interaction* (in press).

S. J. Payne and T. R. G. Green, "Task-Action Grammars: A Model of the Mental Representation of Task Languages," *Human Computer Interaction* **2**, 93-133 (1986).

S. J. Payne and T. R. G. Green, "The Structure of Command Languages: An Experiment on Task-Action Grammars," *International Journal of Man-Machine Studies* **30**, 213-234 (1989).

P. Reisner, "Formal Grammar and Human Factors Design of an Interactive Graphics System," *IEEE Transactions on Software Engineering* **7**, 229-240 (1981).

T. L. Roberts, "Text Editors," in *Handbook of Human-Computer Interaction*, M. Helander, ed. (Elsevier Science Publishers B. V., Amsterdam, North-Holland, 1988).

C. Rouff, *Specification and Rapid Prototyping of User Interfaces,* Ph.D. thesis, University of Southern California, 1991.

A. Schmidt-Nielsen and P. L. Ackerman, "Acquiring Computer Skills: Individual Differences in Style and Ability," Poster presented at the annual meeting of the Human Factors and Ergonomics Association, Seattle, WA, 1993.

B. D. Sharratt, "Top-Down Interactive Systems Design: Some Lessons Learnt from Using the Command Language Grammar," Proceedings of the INTERACT '87 Conference on Human Computer Interaction, Amsterdam, The Netherlands: North Holland, 1987, pp. 395-399.

B. Shneiderman, "Multiparty Grammars and Related Features for Defining Interactive Systems," *IEEE Transactions on Systems, Man, and Cybernetics* **12**,(2), 148-154 (1982).

B. Shneiderman, *Designing the User Interface: Strategies for Effective Human-Computer Interaction* (Addison-Wesley, Reading, MA., 1987).

J. L. Sibert, W. D. Hurley, and T. W. Blesser, "An Object-Oriented User Interface Management System," Proceedings of the ACM SIGGRAPH'86, 1986, pp. 259-268.

J. L. Sibert, W. D. Hurley, and T. W. Blesser, " Design and Implementation of an Object-Oriented User Interface Management System," in *Advances in Human Computer Interaction, Vol. 2,* H. R. Hartson, ed. (Ablex, Norwood, NJ, 1988), pp. 175-213.

A. I. Wasserman, "Extending State Transition Diagrams for the Specification of Human-Computer Interaction," *IEEE Transactions on Software Engineering* **11**,(8), 699-713 (1985).

A. I. Wasserman and D. T. Shewmake, "The Role of Prototypes in the User Software Engineering (USE) Methodology," in *Advances in Human Computer Interaction,* H. R. Hartson, ed. (Ablex, Norwood, NJ, 1985), pp. 191-209.

G. Weil, and J. Helfman, "Specification for Animation Using Keyframes (SPANKEY)," AT&T Bell Laboratories internal memorandum, November 30, 1984.

P. D. Wellner, "Statemaster: A UIMS Based on Statecharts for Prototyping and Target Implementation," Proceedings of CHI'89, 1989, pp. 177-182.

J. Wilson and D. Rosenberg, "Rapid Prototyping for User Interface Design," in *Handbook of Human-Computer Interaction*, M. Helander, ed. (Elsevier Science Publishers B. V., Amsterdam, North-Holland, 1988).

APPENDIX A

UAN SYMBOLS USED IN THE SigmaPlot™ graphing task SPECIFICATION

UAN SYMBOLS FOR THE USER ACTIONS COLUMN

What is Represented	UAN Symbols	Meaning
Cursor movement	~	move the cursor
Object context	[X]	the context of object X, the "handle" by which X is manipulated
Cursor movement	~[X]	move the cursor into the context of object X
Switch operation	v	depress
Switch operation	^	release
String value	K(xyz)	enter value for variable xyz via keyboard
Grouping	( )	grouping mechanism
Sequence	A B	tasks A and B are performed in order left to right, or top to bottom
Repetition	A^n	task A is performed exactly n times
Choice	|, OR	choice of tasks (used to show alternative ways to perform a task)
Specific object value	x'	used to denote a specific object *x* as opposed to any x.
† Keyboard navigation	↑↓, ←→, Tab	use the arrow or Tab keys to navigate the dialogue box, menu, or field
† Location	@	specifies location
† Location	>	"to", in keyboard navigation the object context which follows this symbol is the destination
† Recall	R	recall of a cued key command
† Recall	**R**	recall of an uncued key command
† Null cell		used in cases where there is no equivalent to the action represented in the adjacent cell
† Specific terms	*Ital. Times*	The italicized Times font is used denote terms specific to the application not UAN (e.g., menu names).

†: The dagger denotes new UAN symbols suggested in this paper.

UAN SYMBOLS FOR THE INTERFACE COLUMNS

What is Represented	UAN Symbols	Meaning
Highlight	`!`	highlight object
Unhighlight	`-!`	unhighlight object
Highlight	`!!`	same as `!`, but use a different highlight
Display	`display(X)`	display object X
Erase	`erase(X)`	erase object X
Redisplay	`redisplay(X)`	redisplay object X
For all	$\forall$	for all (e.g., $\forall$icons)
Repetition	A^*	task A is performed zero or more times
† Default	`default=`	(used in the Interface State column) to denote the default value/field of the specified menu, or dialogue box
† Specific terms	*Ital.. Times*	The italicized Times font is used denote terms specific to the application not UAN (e.g., menu names).

†: The dagger denotes new UAN symbols suggested in this paper.

APPENDIX B

COMPLETE MODIFIED SPECIFICATION OF THE SigmaPlot™ graphing task

The following pages contain the complete UAN specification of the **Sigma Plot™ graphing task** in which the user plots four graphs on a page and saves the page to a file. The first table entitled **Sigma Plot™ graphing task** is the task-level description. The actions specified in the User Actions column of this table are macros which are defined at the articulatory level in the pages that follow the task description.

TASK: **SigmaPlot™ graphing task**		
USER ACTIONS	**INTERFACE STATE**	**CONNECTION TO COMPUTATION**
(select graph to plot w/o using F-keys | select graph to plot by clicking graph | select graph to plot using Ctrl-F1	selected= file' & graph'	graph' in file' selected to plot
select plot w/o using F-keys | select plot using Shift F-1	selected= file' & graph' & plot'	plot' selected to plot on graph' in file'
pick data to plot	selected= file' & graph' & plot' & column' column' plotted on graph': selected= ∅ selected= file' & graph'	data from column' plotted as plot' on graph' in file'
select plot w/o using F-keys | select plot using Shift F-1	selected= file' & graph' & plot'	plot' selected to plot on graph' in file'
pick data to plot)[4]	selected= file' & graph' & plot' & column' column' plotted on graph': selected= ∅ selected= file' & graph' selected= ∅	data from column' plotted as plot' on graph' in file'
save file	selected= ∅	updated file' saved
open file | open file by typing in name	selected= file'	file' selected to edit

TASK: **select graph to plot w/o using F-keys**					
USER ACTIONS			**INTERFACE FEEDBACK**	**INTERFACE STATE**	**CONNECTION TO COMPUTATION**
mouse	**keyboard**				
~[*graph* menu]@ menu-barMv	R: Alt-G v^		∀menu label-! *graph* menu label! display (*graph* menu)	selected= *graph* menu	*graph* functions selected
M^			*select graph* menu choice!	default= *select graph* function	
~[*select graph*]@*graph* menu Mv	(↑↓)* > [*select graph* menu choice]	S v^	*select graph* menu choice! display (*select graph* sub-menu)	selected= *graph* menu & *select graph* sub-menu	*select graph* function selected
M^	Enter v^		(selected graph)!	default= *select graph* function & (selected graph)	
~[*graph*']@*select graph* sub-menu Mv	(↑↓)* > [*graph*']	K(graph #') v^	∀ *graph*-! *graph*'!	default= ∅ default= *graph*'	
M^	Enter v^		erase (*select graph* sub-menu) erase (*graph* menu)	selected= ∅ selected= *file*' & *graph*'	*graph*' in *file*' selected to plot

TASK: **select graph to plot by clicking graph**				
USER ACTIONS		**INTERFACE FEEDBACK**	**INTERFACE STATE**	**CONNECTION TO COMPUTATION**
mouse	**keyboard**			
~[*graph*']				
(Mv^)^2		∀*graph*-! *graph*'!	selected= *file*' & *graph*'	*graph*' in *file*' selected to plot

TASK: **select graph to plot using Ctrl-F1**					
USER ACTIONS			**INTERFACE FEEDBACK**	**INTERFACE STATE**	**CONNECTION TO COMPUTATION**
mouse	**keyboard**				
	R: Ctrl-F1 v^		display (*select graph* dialogue box) *graphs available* box! (selected graph)!	selected= *select graph* function default= *graphs available* box default= (selected graph)	*select graph* function selected
	Enter v^	G v^	*graphs available* box!! (selected graph)!!		
~[*graph name*'] @ *graphs available* box Mv	(↑↓)* > [*graph name*']		∀ *graph*-! *graph*'!!		
M^	Enter v^		*graphs available* box-! *graph*'!	default= ∅ default= *graph*'	
~[*OK* box] @*select graph* dialogue box Mv	OR(↑↓,←→,Tab)* > [*OK* box]	O v^	*OK* box!		
M^	Enter v^		erase (*select graph* dialogue box)	selected= ∅ selected= *file*' & *graph*'	*graph*' in *file*' selected to plot

<table>
<tr><td colspan="6">TASK: select plot w/o using F-keys</td></tr>
<tr><th colspan="3">USER ACTIONS</th><th rowspan="2">INTERFACE FEEDBACK</th><th rowspan="2">INTERFACE STATE</th><th rowspan="2">CONNECTION TO COMPUTATION</th></tr>
<tr><th>mouse</th><th colspan="2">keyboard</th></tr>
<tr><td>~[plot menu]@menu-bar Mv</td><td colspan="2" rowspan="2">R: Alt-P v^</td><td>∀ menu label-! plot menu label! display (plot menu)</td><td>selected= plot menu</td><td rowspan="2">plot functions selected</td></tr>
<tr><td>M^</td><td>select plot menu choice!</td><td>default= select plot function</td></tr>
<tr><td>~[select plot] @plot menu Mv</td><td>(↑↓)* > [select plot menu choice]</td><td rowspan="2">S v^</td><td>∀ menu choice-! select plot menu choice! display (select plot sub-menu)</td><td>selected= plot menu & select plot sub-menu</td><td rowspan="2">select plot function selected</td></tr>
<tr><td>M^</td><td>Enter v^</td><td>(selected sub-menu choice)!</td><td>default= select graph function & (selected plot)</td></tr>
<tr><td>~[plot'] @select plot sub-menu Mv</td><td>(↑↓)* > [plot']</td><td rowspan="2">K(plot #') v^</td><td>∀ plot-! plot'!</td><td>default= ∅ default= plot'</td><td></td></tr>
<tr><td>M^</td><td>Enter v^</td><td>erase (select plot sub-menu) erase (plot menu)</td><td>selected= ∅ selected= file' & graph' & plot'</td><td>plot' selected to plot on graph' in file'</td></tr>
</table>

TASK: **select plot using Shift-F1**					
USER ACTIONS			**INTERFACE FEEDBACK**	**INTERFACE STATE**	**CONNECTION TO COMPUTATION**
mouse	**keyboard**				
[redacted]	**R:** Shift-F1 v^		display (*select plot* dialogue box) *plots available* box! (selected plot)!	selected= *select plot* function default= (selected plot)	*select plot* function selected
~[*plot'*]@*plots available* box Mv	Enter v^	P v^	*plots available* box!! (selected plot)!!		
	(↑↓)* > [*plot'*]	K(1st letter of plot') v^	∀ *plot*-! *plot'*!!		
M^	Enter v^		*plots available* box-! *plot'*!	default= ∅ default= *plot'*	
~[*OK* box]@*select plot* dialogue box Mv	OR(↑↓,←→,Tab)* > [*OK* box]	O v^	*OK* box!		
M^	Enter v^		erase (*select plot* dialogue box	selected= ∅ selected= *file'* & *graph'* & *plot'*	*plot'* selected to plot on *graph'* in *file'*

TASK: **pick data to plot**						
USER ACTIONS				**INTERFACE FEEDBACK**	**INTERFACE STATE**	**CONNECTION TO COMPUTATION**
mouse	**keyboard**					
~[*plot* menu]@menu-barMv	R: Alt-P v^			∀menu label-! *plot* menu label! display [*plot* menu]	selected= *plot* menu	*plot* functions selected
M^				*select plot* menu choice!	default= *select plot* function	
~[*pick data to plot*]@*plot* menu Mv	(↑↓)* > [*pick data to plot*]	P v^	R:Shift-F3 v^	∀menu choice-! *pick data to plot* menu choice!	selected= *plot* menu & *pick data to plot* function	*pick data to plot* function selected
M^	Enter v^			display [*pick data to plot* dialogue box] *pairwise XY* box!	default= *select plot function* & *pairwise XY* box	
~[*worksheet* box] @*pick data to plot* dialogue box Mv	OR(↑↓,←→,Tab)* > [*worksheet* box]		W v^	∀box-! *worksheet* box!		
M^	Enter v^			erase (*pick data to plot* dialogue box) erase (*graph page*) display (*worksheet*) (selected column)!	default= ∅ default= (selected column)	
~[*column*']Mv	(←→)* > [*column*'] v^			∀ *column*-! *column*'!	default= ∅ default= *column*'	
M^	Enter v^			display (*column label*)	selected= ∅ selected= *file*' & *graph*' & *plot*' & *column*'	*column*' selected for *plot*'
	Escape v^			*column*'-! erase (*worksheet*) redisplay (*graph page*) plot *column*' > *graph*'	selected= ∅	data from *column*' plotted as *plot*' on *graph*' in *file*'

<table>
<tr><td colspan="7">TASK: **save file**</td></tr>
<tr><th colspan="4">USER ACTIONS</th><th rowspan="2">INTERFACE FEEDBACK</th><th rowspan="2">INTERFACE STATE</th><th rowspan="2">CONNECTION TO COMPUTATION</th></tr>
<tr><th>mouse</th><th colspan="3">keyboard</th></tr>
<tr><td>~[file menu]@menu-bar Mv</td><td colspan="2" rowspan="2">R: Alt-F v^</td><td rowspan="4">R: Ctrl-S v^</td><td>∀ menu label-!
file menu label!
display (file menu)</td><td>selected= file menu</td><td rowspan="2">file functions selected</td></tr>
<tr><td>M^</td><td>new menu choice!</td><td>default= new function</td></tr>
<tr><td>~[save]@file menu Mv</td><td>(↑↓)*
>
[save menu choice]</td><td rowspan="2">S v^</td><td>save menu choice!</td><td>selected= ∅
selected= save function</td><td></td></tr>
<tr><td>M^</td><td>Enter v^</td><td>erase (file menu)
display (saving data message box)
saving finished: erase (saving data message box)</td><td>selected= ∅</td><td>updated file' saved</td></tr>
</table>

TASK: **open file**

USER ACTIONS					INTERFACE FEEDBACK	INTERFACE STATE	CONNECTION TO COMPUTATION
mouse		keyboard					
~[*file* menu]@menu-bar Mv		R: Alt-F v^		R: F2 v^	∀menu label-! *file* menu label! display (*file* menu) *new* menu choice!	selected= *file* menu	*file* functions selected
M^						default= *new function*	
~[open]@file menu Mv		(↑↓)* >[*open* menu choice]	O v^		∀menu choice-! *open* menu choice!	selected= *open* function	*open file* function selected
M^		Enter v^			erase (*file* menu) erase (*graph page*) display (*open file* dialogue box) *path* box!!	default= *path* box	
		Enter v^			*path* box!		
~[*files* box] @*open file* dialogue box Mv	~[*file name*'] @*files* box@*open file* dialogue box Mv	OR(↑↓,←→,Tab)* > [*files* box]	F v^		∀box-! *files* box!	default= *files* box	
M^		Enter v^			*files* box!! (top file name)!	selected= *files* box default=(top file)	
~[*file name*'] @*files* box Mv		(↑↓)* > [*file name*']			∀*file name*-! *file name*'!!	default= ∅ default= *file*'	
M^		Enter v^			*files* box! *file name*'!		
~[*OK* box]@*open file* dialogue box Mv		OR(↑↓,←→,Tab)* > [*OK* box]	O v^		*OK* box!		
M^		Enter v^			erase (*open file* dialogue box) redisplay (*graph page*) display (*file*')	selected= ∅ selected= *file*'	*file*' selected to edit

TASK: **open file by typing in name**

USER ACTIONS				INTERFACE FEEDBACK	INTERFACE STATE	CONNECTION TO COMPUTATION
mouse	**keyboard**					
~[*file* menu]@menu-bar Mv	R: Alt-F v^			∀ menu label-! *file* menu label! display (*file* menu)	selected= *file* menu	*file* functions selected
M^				*new* menu choice!	default= *new function*	
~[open]@file menu Mv	(↑↓)* >*[open* menu choice]		R: F2 v^	∀ menu choice-! *open* menu choice!	selected= *open* function	*open* graph file function selected
M^	Enter v^	O v^		erase (*file* menu) erase (*graph page*) display (*open file* dialogue box) *path* box!!	default= *path box*	
	K(file name ')			*display= (file name)*		
~[OK box]@open file dialogue box Mv	*Enter v^*			*OK box!*		
M^				erase (*open file* dialogue box) redisplay (*graph page*) display (*file* ')	selected= Ø selected= *file* '	*file* ' selected to edit